IT 314

Software Engineering

RESTAURANT CONTROLLER UNIT TESTING

Fork & Feast

Group 28

➢ Tool Used:

Mocha @10.8.2
Chai @4.3.4
Other – Sinon @19.0.2

Link to the source code: [restaurantController]

1. Add Restaurant Validation

This function handles the creation of a new restaurant entry in the database. It extracts and processes data from the incoming request, including mapping file uploads for images, parsing the capacity field, and assembling the restaurant's details into a new instance of the restaurant model. Upon successfully saving the entry, it returns a 201 status code with a success message and the newly created restaurant; in case of an error, it logs the error and responds with a 401 status code and an error message.

a) Should return 400 if both images not provided

```
it('should return 400 if both images are not provided', async () => {
    // Arrange
    req.files.image = undefined; // No image provided
    req.files.menuImage = undefined; // No menu image provided

    // Act
    await addRestaurant(req, res);

    // Assert
    expect(res.status.calledWith(400)).to.be.true;
    expect(res.json.calledWith({ message: 'Images are required' })).to.be.true;
});
```

It simulates a scenario where neither image nor menuImage files are provided in the request. The addRestaurant function is called with the mocked req and res objects. The test asserts that res.status is called with 400 and res.json returns a message indicating that images are required.

b) Should Validate Opening/Closing Time

```
it('should properly validate edge case times', async () => {
    const testCases = [
        { openTime: '00:00', closeTime: '23:59', shouldBeValid: true },
        { openTime: '23:59', closeTime: '00:01', shouldBeValid: true },
        { openTime: '12:00', closeTime: '12:00', shouldBeValid: false },
        { openTime: '23:59', closeTime: '23:58', shouldBeValid: false }
    ];

    for (const testCase of testCases) {
        req.body.openingTime = testCase.openTime;
        req.body.closingTime = testCase.closeTime;

        const saveStub = sinon.stub(restaurant.prototype, 'save').resolves({
            ...req.body,
            _id: 'testRestaurantId'
        });

        await addRestaurant(req, res);

        if (testCase.shouldBeValid) {
            expect(res.status.calledWith(201)).to.be.true;
            expect(saveStub.calledOnce).to.be.true;
        } else {
            expect(res.status.calledWith(400)).to.be.true;
            expect(res.json.calledWith({
                message: "Invalid Time Range"
            })).to.be.true;
        }

        sinon.restore(); // Reset stubs for next iteration
    }
});
```

It tests the validation of edge case opening and closing times for a restaurant. For each test case, it sets openingTime and closingTime in the request and stubs the save method of the restaurant model. The addRestaurant function is called, and the test asserts that valid time ranges result in a 201 status and the save method being called, while invalid ranges return a 400 status with an "Invalid Time Range" error message. The stub is restored after each iteration.

```
addRestaurant validation
  ✓ should return 400 if both images are not provided
  ✓ should properly validate edge case times
```

These both test cases are successfully covered which is confirmed with the result shown in the terminal.

c) Should Handle Malformed Timings

```
it('should handle malformed time strings', async () => {
    const invalidTimeCases = [
        { openTime: '', closeTime: '17:00' },
        { openTime: '09:00', closeTime: '' },
        { openTime: 'invalid', closeTime: '17:00' },
        { openTime: '09:00', closeTime: 'invalid' },
        { openTime: '25:00', closeTime: '17:00' },
        { openTime: '09:00', closeTime: '24:01' }
    ];

    for (const testCase of invalidTimeCases) {
        req.body.openingTime = testCase.openTime;
        req.body.closingTime = testCase.closeTime;

        await addRestaurant(req, res);

        expect(res.status.calledWith(400)).to.be.true;
        expect(res.json.calledWith({
            message: "Invalid Time Range"
        })).to.be.true;

        sinon.restore(); // Reset stubs for next iteration
    }
});
```

It tests the handling of malformed time strings in openingTime and closingTime. For each invalid case, it sets the times in the request, calls the addRestaurant function, and asserts that the response includes a 400 status and an "Invalid Time Range" error message. Stubs are reset after each iteration.

This test case is successfully covered which is confirmed by the result shown in terminal:

```
✓ should handle malformed time strings (46ms)
```

2. All Restaurant Show Validation

The allRestaurant function retrieves all restaurants for the logged-in owner by filtering based on the ownerId from the request. It transforms the restaurant data to include only the first image for each restaurant and responds with the modified data and a 201 status. In case of an error, it logs the error and responds with a 401 status and an error message.

a) Returning all restaurants owned by the owner

```
it('should return all restaurants for owner', async () => {
    const mockRestaurants = [
        {
            _id: 'rest1',
            name: 'Restaurant 1',
            image: ['image1.jpg', 'image2.jpg'],
            toObject: function() {
                return {
                    _id: this._id,
                    name: this.name,
                    image: this.image
                };
            }
        }
    ];

    sinon.stub(restaurant, 'find').resolves(mockRestaurants);

    await allRestaurant(req, res);

    expect(res.status.calledWith(201)).to.be.true;
    expect(res.json.calledWith({
        restaurantData: sinon.match.array
    })).to.be.true;
});
```

It tests retrieving all restaurants for a specific owner. A mock restaurant list is stubbed using the find method, simulating a database response. The

allRestaurant function is called, and the test asserts that the response includes a 201 status and a JSON object with an array of restaurant data.

b) Handling errors (related to Database)

```
it('should handle errors', async () => {
    sinon.stub(restaurant, 'find').rejects(new Error('Database error'));

    await allRestaurant(req, res);

    expect(res.status.calledWith(401)).to.be.true;
    expect(res.json.calledWith({
        error: 'Database error'
    })).to.be.true;
});
```

It tests error handling when retrieving restaurants. The find method is stubbed to throw a "Database error." The allRestaurant function is called, and the test asserts that the response includes a 401 status and a JSON object with the error message "Database error.

These both test cases are successfully covered which is confirmed with the result shown in the terminal.

```
allRestaurant
  ✓ should return all restaurants for owner
✓ should handle errors
```

3. Get Restaurant by ID

The GetRestaurantById function retrieves a restaurant by its ID from the database. If the restaurant is not found, it responds with a 404 status and a "Restaurant Not Found" message. On success, it returns the restaurant data with a 200 status. In case of an error, it logs the error and responds with a 401 status and the error message.

a) Returning restaurant by giving ID

```
it('should return restaurant by id', async () => {
    const mockRestaurant = {
        _id: 'testRestaurantId',
        name: 'Test Restaurant'
    };

    req.params.id = 'testRestaurantId';
    sinon.stub(restaurant, 'findById').resolves(mockRestaurant);

    await GetRestaurantById(req, res);

    expect(res.status.calledWith(200)).to.be.true;
    expect(res.json.calledWith({
        restaurantData: mockRestaurant
    })).to.be.true;
});
```

It tests retrieving a restaurant by its ID. A mock restaurant is stubbed to simulate the findById method returning a valid restaurant. The GetRestaurantById function is called, and the test asserts that the response includes a 200 status and a JSON object with the restaurant data.

b) Handling Non-existent Restaurants

```
it('should handle non-existent restaurant', async () => {
    req.params.id = 'nonexistentId';
    sinon.stub(restaurant, 'findById').resolves(null);

    await GetRestaurantById(req, res);

    expect(res.status.calledWith(404)).to.be.true;
    expect(res.json.calledWith({
        message: 'Restaurant Not Found'
    })).to.be.true;
});
```

It tests handling a non-existent restaurant ID. The findById method is stubbed to return null to simulate a missing restaurant. The GetRestaurantById function is called, and the test asserts that the response includes a 404 status and a JSON object with the message "Restaurant Not Found."

c) Handling exception and returning error

```
it('should return 401 and error message when an exception is thrown', async () => {
    const error = new Error('Database connection error');
    sinon.stub(restaurant, 'findById').throws(error);

    await GetRestaurantById(req, res, next);

    expect(res.status.calledOnceWith(401)).to.be.true;
    expect(res.json.calledOnceWith({ error: 'Database connection error' })).to.be.true;
});
```

It tests error handling when an exception occurs during the retrieval of a restaurant by ID. The findById method is stubbed to throw a "Database connection error." The GetRestaurantById function is called, and the test asserts that the response includes a 401 status and a JSON object with the error message “Database connection error”.

```
GetRestaurantById
  ✓ should return restaurant by id
  ✓ should handle non-existent restaurant
✓ should return 401 and error message when an exception is thrown
```

These test cases are successfully covered which is confirmed with the result shown in the terminal.

4. Update Restaurant

The updateRestaurant function updates a restaurant's details for the logged-in owner. It retrieves the restaurant by ID and verifies ownership. If the restaurant is found, it updates its fields with the new data from the request, including parsing capacity and mapping uploaded images. The updated restaurant is saved, and a 200 status with a success message and updated data is returned. If not found or unauthorized, it responds with a 404 status, and any errors are logged and responded to with a 401 status.

a) Should successfully update restaurant details

```
it('should successfully update restaurant', async () => {
    const mockRestaurant = {
        _id: 'testRestaurantId',
        ...req.body,
        save: sinon.stub().resolves()
    };

    sinon.stub(restaurant, 'findOne').resolves(mockRestaurant);

    await updateRestaurant(req, res);

    expect(res.status.calledWith(200)).to.be.true;
    expect(res.json.calledWith({
        message: 'Restaurant updated successfully',
        restaurant: sinon.match.object
    })).to.be.true;
});
```

It tests successfully updating a restaurant. A mock restaurant is stubbed to simulate the findOne method returning a matching restaurant, and its save method is stubbed to resolve successfully. The updateRestaurant function is called, and the test asserts that the response includes a 200 status and a JSON object with a success message and the updated restaurant.

b) Should handle error

```
it('should handle errors and return a 401 status', async () => {
    // Arrange
    const errorMessage = 'Database error';

    // Stub the findOne method to resolve a restaurant
    sinon.stub(restaurant, 'findOne').resolves({
        _id: 'testRestaurantId',
        ...req.body,
        save: sinon.stub().rejects(new Error(errorMessage)) // Simula
    });

    // Act
    await updateRestaurant(req, res);

    // Assert
    expect(res.status.calledWith(401)).to.be.true; // Check that res.
    expect(res.json.calledWith({ error: errorMessage })).to.be.true;
});
```

It tests error handling during the restaurant update process. The findOne method is stubbed to return a mock restaurant, but the save method is set to reject with a "Database error." The updateRestaurant function is called, and the test asserts that the response includes a 401 status and a JSON object with the error message “Database error”.

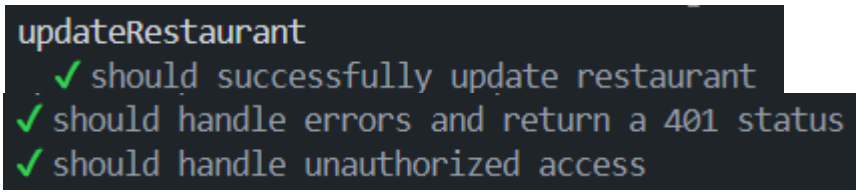
c) Should handle unauthorized access

```
it('should handle unauthorized access', async () => {
    sinon.stub(restaurant, 'findOne').resolves(null);

    await updateRestaurant(req, res);

    expect(res.status.calledWith(404)).to.be.true;
    expect(res.json.calledWith({
        message: 'Restaurant not found or unauthorized access'
    })).to.be.true;
});
```

It tests handling of unauthorized access or when the restaurant is not found. The findOne method is stubbed to return null, simulating a

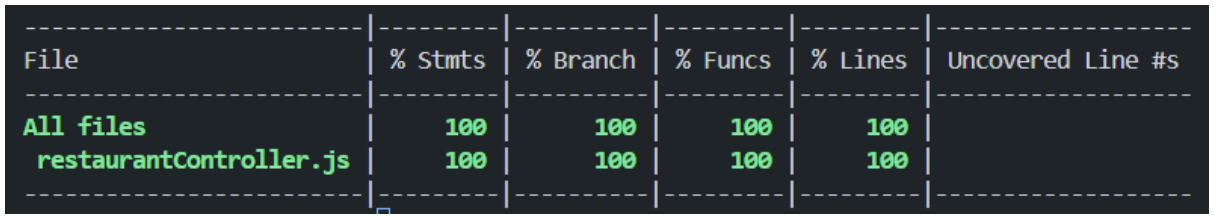
scenario where the restaurant does not exist or the user is unauthorized. The updateRestaurant function is called, and the test asserts that the response includes a 404 status and a JSON object with the message "Restaurant not found or unauthorized access.

These test cases are successfully covered which is confirmed with the result shown in the terminal.



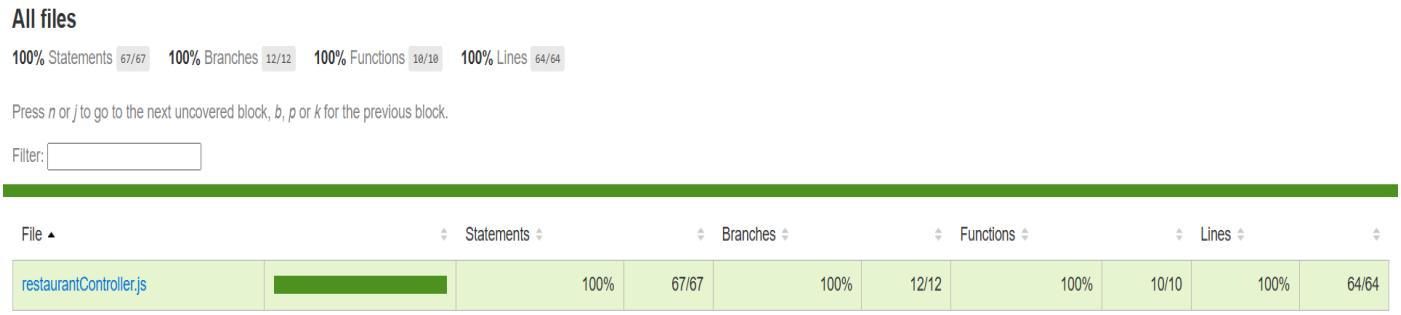
- **Coverage Report**

For this we used the “nyc --reporter=text” command which gave us the
following output:



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
restaurantController.js	100	100	100	100	

For in depth report we used “nyc --reporter=lcov” which gave us an html file highlighting the number of times a particular line ran during the entire testing.



Link to the HTML File: [coverage_report]